Journal of Biomedical Informatics 47 (2014) 62–70

Contents lists available at ScienceDirect

Journal of Biomedical Informatics

journal homepage: www.elsevier.com/locate/yjbin

FALCON or how to compute measures time efficiently on dynamically evolving dense complex networks?

R. Franke *, G. Ivanova

Institute of Computer Science, Humboldt-Universität zu Berlin, Unter den Linden 6, 10099 Berlin, Germany

ARTICLE INFO

Article history:
Received 7 February 2013
Accepted 10 September 2013
Available online 21 September 2013

Keywords:
Complex network
Brain network
OpenCL
GPGPU
Code optimization
SSE

ABSTRACT

A large number of topics in biology, medicine, neuroscience, psychology and sociology can be generally described via complex networks in order to investigate fundamental questions of structure, connectivity, information exchange and causality. Especially, research on biological networks like functional spatio-temporal brain activations and changes, caused by neuropsychiatric pathologies, is promising. Analyzing those so-called complex networks, the calculation of meaningful measures can be very long-winded depending on their size and structure. Even worse, in many labs only standard desktop computers are accessible to perform those calculations. Numerous investigations on complex networks regard huge but sparsely connected network structures, where most network nodes are connected to only a few others. Currently, there are several libraries available to tackle this kind of networks. A problem arises when not only a few big and sparse networks have to be analyzed, but hundreds or thousands of smaller and conceivably dense networks (e.g. in measuring brain activation over time). Then every minute per network is crucial. For these cases there several possibilities to use standard hardware more efficiently. It is not sufficient to apply just standard algorithms for dense graph characteristics. This article introduces the new library *FALCON* developed especially for the exploration of dense complex networks. Currently, it offers 12 different measures (like clustering coefficients), each for undirected-unweighted, undirected-weighted and directed-unweighted networks. It uses a multi-core approach in combination with comprehensive code and hardware optimizations. There is an alternative massively parallel GPU implementation for the most time-consuming measures, too. Finally, a comparing benchmark is integrated to support the choice of the most suitable library for a particular network issue.

1. Introduction

1.1. Networks are everywhere

Many systems in nature, society and technology can be described by networks or, more mathematically, graphs. The study of those so-called complex networks is an interdisciplinary field which deals with general structural properties in biological, neural, physical, chemical, social or technical network structures [1–4]. Famous examples are the analysis of huge social networks consisting of millions of people connected by their acquaintance [5] and protein interaction networks [6]. Interestingly, networks of different fields show comparable characteristics like very sparse connectivity or the small-world property as there exist very short paths between arbitrarily chosen nodes [2].

A complex network abstracts from a certain topic and provides uniform measures to analyze inherent network properties. As a common basis mathematical graph theory is used, which treats networks as a set of nodes connected by (un)directed and (un)-weighted edges (also called links) regardless of the context they represent. For example, a weight can symbolize a strength, length or intensity in a specific context. In order to guarantee the applicability of some measure formulas, these complex networks are restricted to edge weights between 0 and 1, no self-connections of nodes and not more than one edge from one node to another.

Although complex networks are inherently general, they reflect biological structures in nature on many scales. Researches investigated protein interaction networks [8], epidemic spreading [9,10], cellular networks [11], protein folding [12], metabolic networks [13] and many more. In medicine, psychology and neuroscience, complex networks are used to analyze normal and pathological brain structures and spatiotemporal dynamics of brain activation [14], learning processes [15], early human brain development [16], resting state networks [17,18], cognitive state changes [19], age and sex differences [20,21] and neuropsychological differences

* Corresponding author. Tel.: +49 30 2093 5488.
E-mail addresses: rfranke@informatik.hu-berlin.de (R. Franke), givanova@informatik.hu-berlin.de (G. Ivanova).

1532-0464/$ - see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jbi.2013.09.005

in Alzheimer's disease [22], ADHD [23], schizophrenia [24], epilepsy [25] and other pathologies. An overview of the study of psychopathology with complex networks can be found in [26].

A comprehensive overview about complex networks, their measures and formulas can be found in [27]. The formulas given there were the theoretical basis for this article. Whereas most measure formulas use sums and products over nodes and edges, some of them need search algorithms on graph data structures to find, e.g. shortest path lengths between node pairs. Technically seen, there are different classical data structures which represent graphs (see Fig. 1). Since there are also different standard search algorithms, the choice and optimization of a suitable algorithm as well as data structure is of a high importance.

1.2. Different networks in science

On the one hand, the popular social networks can get very large with several millions of nodes, but mostly the edges are distributed very sparsely between them [5], since not everybody knows thousands of other people. The edge density, the fraction of existing edges ($\frac{e}{n^2-n}$ with n nodes and e directed edges or $e/\frac{n^2-n}{2}$ with e undirected edges), is exceptionally low. An often observed characteristic is described as small-world property. In that case, most nodes are only sparsely connected, but every node has only a very short path to all the others [2,4]. In this social network context analysis can concentrate on one or a few of those networks. In that case, algorithms and data structures for sparse networks will be applied, like Dijkstra's algorithm for weighted shortest path lengths on adjacency lists (see Fig. 1).

On the other hand, one could for example investigate functional, causal and effective connections inside the human brain [27]. A representative example would be the exploration of EEG signals, acquired during brain stimulation or rest. Using methods for estimation of distributed brain sources and their temporal connectivity, a network of connected sources for each time step can be built. Such a source network would preferably represent a high resolution of several thousand sources (nodes). Then it is possible to investigate the spatiotemporal relation of integration and localization of information in the brain [28]. Since this is explorative research, it is not clear how much information (lightly weighted edges) can be left out, so that, in the computationally worst case, a network could be fully connected with weighted edges. In other words, the edge density would be near 100%. It is obvious, that this problem can occur in every approach, where connections between nodes are calculated statistically from measured data.

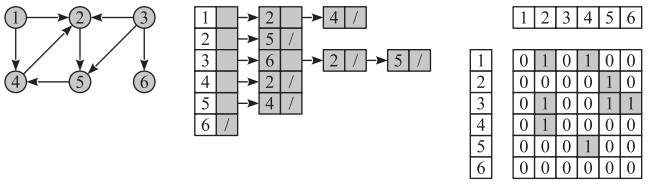

Fig. 1. Graph representations. This figure shows a directed, unweighted network on the left and two standard graph representations for it. The figure is based on [7]. In the center one can see an adjacency list, which stores edges for every node and therefore only needs as much memory as edges and nodes exist, which is important for networks with millions of nodes but relatively few edges. On the right, an adjacency matrix is displayed that stores edges in a table which occupies n^2 entries if n is the number of nodes. Even if there is no edge, a zero value must be stored. The advantage of such a matrix in comparison to a list is the fast access to its elements and less memory requirement for very dense networks. The disadvantage is the waste of memory for sparsely connected networks. An undirected network only needs half of the matrix. If the network is weighted, every edge in the list needs an additional weight value to be stored. In the matrix, elements just become weight values, whereby a weight of 0 symbolizes no edge. Most graph algorithms can be applied on both lists or matrices.

1.3. A problem of time

If a lot of (maybe dense) networks have to be analyzed, calculation runtime becomes crucial. Let us assume, we want to examine the temporal course of brain network properties over time in a simple EEG experiment. Then for every time step a brain network has to be constructed and network measures have to be calculated. When for example investigating fine-grained, dense networks of 5000 nodes (e.g. distributed sources as nodes) with weighted edges (e.g. temporal source correlations), calculation of even fundamental measures can take a few minutes on a standard desktop PC. It depends on used hardware, algorithms, graph data structures, implementation and edge density. When we want to know all clustering coefficients and all shortest path lengths in those networks then this may take perhaps 20 min per network.

For one network, this may be just annoying to wait for. But if we analyze an experiment of only 10 s and create a network every 50 ms for a good temporal resolution, than 2000 networks have to be investigated (each with 5000 nodes and at worst up to almost 25,000,000 weighted edges). Even ignoring each network creation, a successive measure calculation would take about 28 days for one trial of this experiment.

For those cases, the need for faster computational approaches for dense networks on standard hardware is obvious. Unfortunately, just the most fundamental measures, that serve as basis for higher measures, are the hardest to calculate, as shown in the next section.

1.4. Hierarchy of measures

The hierarchy diagram (Fig. 2) shows computational dependencies between a few complex network measures, each computed for all nodes, node pairs or the network. The corresponding formulas can be found in [27]. Every measure needs its predecessor measures to be computed before itself can be retrieved. Thus, it inherits the computational complexity of its predecessors. Therefore it is the most important task to optimize the fundamental computations like the numbers of (undirected, weighted or directed) triangles around all nodes and the shortest (undirected, weighted or directed) path lengths between all node pairs. Unfortunately, both need the most time since they have a cubic time complexity in the worst-case when calculated for all nodes.

The denser a network is, the more calculation actually becomes this worst-case scenario. Since there is a lack of special treatment for this kind of networks, this causes serious time problems as mentioned above. So, it became necessary to create a highly optimized, but general complex network analysis software for standard hardware to enable valuable time saving for current research.

2. FALCON – a new library for dense networks

FALCON (FAst Library for COmplex Networks) was developed because of the lack of network analysis software specialized in dense complex network analysis. The kind of EEG experiments described in the last chapter demonstrates one of the primary reasons to develop a new optimized library. The goal of *FALCON* was to compute all measures shown in Fig. 2, each for all nodes, node pairs respectively the entire network.

There are three edge types: undirected (unweighted), (undirected) weighted, and directed (unweighted) edges. Node-based measures for these three versions are betweenness centralities, numbers of triangles, average neighbor degrees, clustering

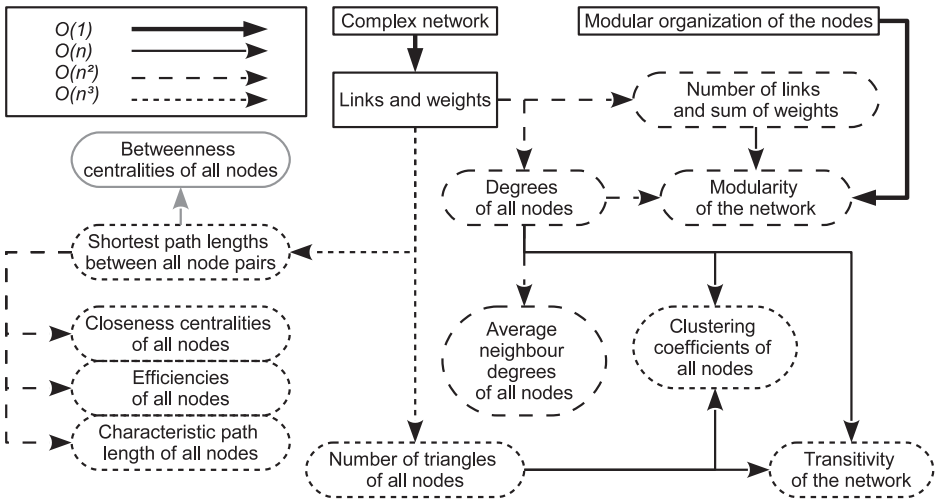

Fig. 2. Measure dependencies and corresponding worst-case runtimes. This diagram illustrates the dependencies and time complexities that exist between individual measures and their computation for all nodes in a network. It was derived from [27]. n is the number of nodes. Every measure is calculated from its predecessors. For example, the transitivity (right bottom corner) depends on degrees and numbers of triangles. The dotting of lines indicate the computational time complexity. A thick and straight line represents constant time, a thin and straight line symbolizes linear time, a dashed, thin line quadratic and the dotted line cubic time complexity. The dotting of an arrow represents the calculation time needed from one measure to the next. The dotting of an oval (measure) or square (precondition) indicates the overall calculation time from a network to this measure. That means that the transitivity is calculated in quadratic time out of degrees and numbers of triangles, but altogether, it needs asymptotic cubic time because the numbers of triangles is that expensive. The gray arrow from shortest path lengths to betweenness centralities only indicates a computational relationship but they are in fact computed by a separate algorithm (Brandes' algorithm, [29,30]). Please note that there is an undirected, weighted and directed version of each measure, except for degrees (6 versions).

coefficients, average distances, efficiencies and closeness centralities. Based on node pairs are shortest path lengths. For the whole network there are number of links respectively sum of weights, transitivity and modularity. At last, there are six kinds of degrees for all nodes: (undirected, unweighted) degrees, directed degrees, directed reciprocal degrees, directed in-degrees, directed out-degrees and weighted degrees.

Measures for directed, weighted edges are issue of current research and not implemented yet. But there are many of them available in the *Brain Connectivity Toolbox (BCT)* [27].

2.1. Methods

The most important design consideration when developing *FALCON* was an effective utilization of available desktop computer resources in order to optimize calculation speed. This includes the following techniques.

Parallelization on multiple cores – The amount of data to be processed is distributed to a given number of processor cores. The user is free to choose the number of threads (preferably the number of CPU cores). Because even the higher measure formulas contain simple sums and products over data, it is mostly possible to achieve a good scaling which means that p parallel processes are almost p times faster than just one process.

Streaming SIMD extensions – On Intel and AMD processors Streaming SIMD Extensions (SSE) enable performing one operation on multiple data at a time. SIMD stands for Single Instruction Multiple Data. Applied on an array of data like edge weights this can improve performance by a factor of x (usually $2, 4, \ldots$) if the SSE instruction handles x values. Since the most measures contain only fundamental arithmetic operations and most of the sums and products can be calculated in a successive way, SSE instructions can be applied. To use SSE, array memory representations have to be aligned on 16 bytes and treated carefully then used in two dimensions of adjacency matrices.

Efficient memory and cache utilization – In fact, working memory is very much slower than the CPU calculates. Only fast cache memory in between enables fast processing. A cache assumes temporal and spatial locality of data accesses. So, data should be accessed and reused in a successive way, which can be done by modifying loops over data. If useful and not automatically done by the compiler, the applied techniques were: interchanging the order of nested loops (to access 2D data successively in main memory), prefetching of distant data and unrolling loops (to reuse data for calculations as often as possible inside of a cache line and reduce jump instructions in loops), organizing two-dimensional memory accesses in small 2D blocks that fit into cache lines (to reduce the amount of jumps in memory) and adjusting the resulting block sizes to a multiple of the cache line size (to avoid capacity and page misses). When calculating on multiple cores, processes should not interfere to close in their memory accesses to prevent false sharing. This is a time-wasting process of establishing cache coherence in a cache line when several processes are writing on it, even if every process accesses different data.

Algorithmic modifications – This section summarizes all techniques that prevent useless operations by for example checking mathematical conditions. An example is the calculation of the numbers of triangles where an if-statement aborts memory access on further parts of a triangle when the first node partner is not connected and therefore no triangle can be constituted. The difficulty is to determine if checking pays off because it needs time to be done itself. So, it is a trade-off depending on the network structure. Generally, these optimizations are simple and effective on networks with low and medium edge density since they prevent unnecessary operations and jumps in memory.

GPGPU – An upcoming alternative to multi-core CPUs is General Purpose Computation on Graphics Processing Unit (GPGPU). That means calculating on modern graphic cards, which are optimized to process a huge amount of little threads (kernels) in parallel to render geometric primitives. In case of parallelizable algorithms the usage of GPUs for scientific calculations can result in a great performance improvement compared to current CPUs of the same price category. A general interface for GPU-computing used in this study is the Open Computing Language (OpenCL), available for both AMD (ATI Stream technology) [31] and NVIDIA (CUDA architecture) [32] hardware. As for CPU approaches, access in memory blocks with shared memory and unrolling are important ways to improve performance. Those measures with the highest

complexity were implemented with OpenCL. That were numbers of triangles and shortest path lengths each for three edge types. The OpenCL implementation for shortest path lengths was theoretically based on former approaches with CUDA in [33,34] and SDK material from AMD [31], where, e.g. block-wise access patterns were used to reuse data as much as possible and to load parts of graphs bigger than GPU memory. Since *FALCON* focuses on graphs fitting into GPU memory, an overhead of block processing kernel calls was avoided and instead of quadratic blocks shared memory lines appears to be faster in this case.

Combining techniques – The challenge during the implementation was to obtain the best possible optimization and synergy between different techniques. For example, the combination of multi-core processing and SSE instructions needs a very prudent implementation. Extensive information on optimization techniques can be found in [35,36,32,31]. The only algorithms that not just implement the formulas as loops over data are the calculation of shortest path lengths of all node pairs and the betweenness centralities of all nodes. First ones are done by the Floyd–Warshall algorithm as it is the best choice calculating the path lengths for all node pairs in rather dense networks in comparison to several alternatives [7]. This will be explained and shown in the benchmark (Section 3). The betweenness centralities are computed with Brandes' algorithm [29,30].

2.2. Performance has its price

To take advantage of those speed optimizations, adjacency matrices are used to represent networks (Fig. 1). In fact, this is the disadvantage of *FALCON*'s approach. As shown in Fig. 1 these data structures are simple and fast but their size increases quadratically with the number of nodes.

FALCON's link weights and weighted results are based on the float data type (4 bytes) instead of double (8 bytes). Advantages are reduced size and more speed, when using SSE, since it can handle 4 float values at once in contrast to just 2 double values. So, if usable, runtime with floats can be twice as fast as with double.

The price to pay is lower data precision. The data type float has a decimal data precision of 6 digits. For example, 123.4561 and 123.4562 are considered equal as well as 0.0001234561 and 0.0001234562. The double data type offers 15 decimal digits. If such a high resolution is really required, FALCON is not applicable in the current version.

2.3. Memory requirements

The memory requirement (RAM or GPU memory) for an adjacency matrix with n nodes depends on the data type size to store edges. Unweighted edges are currently stored with 1 byte (see Section 4 for improvements) and weighted edges with 4 bytes (float). If b is the number of bytes per edge then an adjacency matrix needs $M = n^2 \cdot b$ bytes in memory, so using double (8 bytes) instead of float doubles the memory needed. In every case, there are $n^2 - n$ possible directed edges (substraction of n is caused by the prohibition of self-connections) or $\frac{n^2-n}{2}$ undirected edges, because only half of the symmetric matrix is used. The other half is either wasted (see Section 4 for improvements) or at best used for speed optimization like in FALCON.

A memory size formula for a generic adjacency list is far more complicated. If list elements have only one pointer to the follower, the list is called single-linked (please see Fig. 1). When elements have an additional pointer back to the predecessor, the list is double-linked. A pointer needs $b_{\rightarrow} = 4$ bytes = 32 bit in a 32-bit program and $b_{\rightarrow} = 8$ bytes in 64-bit programs. In a simple approach one stores one edge list pointer for each of n nodes and then links e edges, each with a target node ID (b_{ID}: 1, 2, 4 or 8 bytes dependent on maximum node ID to store). Please note that 1 byte has 8 bits, thus offers $2^{1\cdot 8} = 256$ possible integer IDs, 2 bytes enable $2^{2\cdot 8} = 65{,}536$ IDs, 4 bytes $2^{4\cdot 8}$ IDs and 8 bytes $2^{8\cdot 8}$ IDs. Every list element needs one or two pointers to neighbor list elements ($n_{\rightarrow} = 1$ or $n_{\rightarrow} = 2$) and an optional weight (b_{w}: 0 bytes for unweighted, 4 bytes for float, 8 bytes for double). Then, one needs to define, how to manage undirected edges. If speed is important, than it is useful to store an edge (a,b) as list element as well as (b,a) just like the matrix would do (scale factor for stored edge number $s_e = 2$). The other possibility is to store it only once, for example with sorted node IDs so that only edges (a,b) with ID $(a) <$ ID (b) are stored ($s_e = 1$). Please note, that a library has to explicitly support suitable features for an adjacency list network like single-linking (C++ standard list is double-linked), smaller node ID data types if possible or storing undirected edges once (and suited algorithms to work with them). An adjacency list then requires $L = n \cdot b_{\rightarrow} + s_e \cdot e \cdot (b_{\mathrm{ID}} + n_{\rightarrow} \cdot b_{\rightarrow} + b_{\mathrm{w}})$ bytes. Please note, that there are ways to implement rather static lists without pointers, but again, libraries have to support this.

For example, let us say we have 2 GB (=2×1024^3 bytes) free memory. If completely reserved for a weighted (and directed or undirected) adjacency matrix, it could have $n \leqslant \sqrt{\frac{M}{b}} = \sqrt{\frac{2\times 1024^3 \text{ bytes}}{4 \text{ bytes}}} \approx 23{,}170$ nodes. Using the maximum of $n = 23{,}170$ nodes there are $e = n^2 - n = 536{,}825{,}730$ possible directed, weighted edges or 268,412,865 undirected, weighted edges. An unweighted (and directed or undirected) matrix could have $n \leqslant \sqrt{\frac{M}{b}} = \sqrt{\frac{2\times 1024^3 \text{ bytes}}{1 \text{ bytes}}} \approx 46{,}340$ nodes and at most 2,147,349,260 directed or 1,073,674,630 undirected edges.

For comparison we take a weighted, undirected adjacency list with features consuming minimal memory. Such a float-weighted ($b_{\mathrm{w}} = 4$ bytes), single-linked ($n_{\rightarrow} = 1$) adjacency list in a 32-bit program ($b_{\rightarrow} = 4$) with undirected edges stored once ($s_e = 1$) could have different numbers of nodes and edges to fill the same memory $L = 2$ GB. For example, with a fixed node number of $n = 40{,}000$ we can store a node ID in a 2 bytes value handling up to 65,536 node IDs ($b_{\mathrm{ID}} = 2$). Then, there could be $e \leqslant \frac{L - n \cdot b_{\rightarrow}}{s_e \cdot (b_{\mathrm{ID}} + n_{\rightarrow} \cdot b_{\rightarrow} + b_{\mathrm{w}})} \approx 214{,}732{,}364$ undirected edges corresponding to a maximal undirected edge density of $e / \frac{n^2-n}{2} \approx 26.8\%$. A double weight ($b_{\mathrm{w}} = 8$) would result in $e \leqslant 153{,}380{,}260$ edges (at most 19.1% density). Using a modern 64-bit program ($b_{\rightarrow} = 8$) would result in maximal densities of 19.1% (float weights) respectively 14.9% (double weights) for $n = 40{,}000$ nodes in 2 GB.

Now, we look at different edge densities by example. A float-weighted, undirected matrix with $n = 40{,}000$ nodes and a low edge density of, e.g. 1% would have $e = 0.01 \cdot \frac{n^2-n}{2} = 7{,}999{,}800$ undirected edges and needs $M \approx 5.96$ GB. The same network with a high edge density of, e.g. 75% would have $e = 0.75 \cdot \frac{n^2-n}{2} = 599{,}985{,}000$ undirected edges and requires again $M \approx 5.96$ GB. In contrast to that, an adjacency list depends on their features. Our example of an memory saving undirected weighted adjacency list ($b_{\rightarrow} = 4$, $s_e = 1$, $b_{\mathrm{ID}} = 2$, $n_{\rightarrow} = 1$, $b_{\rightarrow} = 4$, $b_{\mathrm{w}} = 4$) with the same node number needs only $L \approx 76.4$ MB for 1% density, so the list is saving a lot of memory. For 75% density it would need $L \approx 5.59$ GB in a 32-bit program and $L \approx 7.82$ GB in 64-bit programs. If a typical standard list is used (double-linked, 4 bytes per node ID), it would need 122.22 MB (32-bit) or 183.41 MB (64-bit) for 1%. For 75% it would need 8.94 GB (32-bit) or 13.41 GB (64-bit). Storing undirected edges twice for better access almost doubles needed memory.

Therefore it is not possible to say which of many possible network data structures uses more memory than others unless one includes all parameters. There is no obvious other way to store very dense networks such compactly as in matrices.

Please note, that at no time 100% of main or GPU memory is free available and especially a matrix needs a whole free block of memory. Additionally, when calculating with Floyd–Warshall algorithm one needs values for path lengths of all n^2 nodepairs, each with 4 bytes for float (weighted) or unsigned int (unweighted). All other measure results are only one-dimensional (vectors of length n) or even only one value. Temporary variables are used but not in a way influencing the rough memory requirements.

3. Benchmark

To show that *FALCON* is able to save a lot of time, it is reasonable to compare its runtime performance with other libraries specialized in graph or complex network operations at the same test computer, program and networks. Please note, that all comparisons to FALCON operate on the data precision of float, even when another library uses double internally. All benchmarked libraries used exact algorithms and do not use any approximations or search cutoffs.

3.1. Choice of test measures

There are two reasons to restrict the benchmark for this article to the weighted shortest path lengths of all node pairs and the undirected clustering coefficients of all nodes.[1]

The first reason is that the runtimes needed for calculating all shortest path lengths (no matter if undirected, weighted or directed) and the numbers of triangles around all nodes (required for clustering coefficients) are cubic in the worst case. Since these measures start two main calculation routes (see Fig. 2), their runtime performance is crucial for every successing measure in the hierarchy.

Secondly, general graph libraries do not necessarily support complex network measures, but shortest paths lengths and (undirected) clustering coefficients are widely supported. But even weighted and directed clustering coefficients are not included in every tested library and, even more confusing, there are different weighted generalizations of clustering coefficients (see [37] for an overview). For example, the libraries *networkx* and *igraph* use different formulas. *FALCON* currently implements the weighted clustering coefficient proposed by Onnela et al. [38].

The weighted version of shortest path lengths was investigated, because it is supported with several algorithms by the libraries *BGL*, *BCT*, *igraph* and *networkx*. Furthermore, this version is especially meaningful for very dense networks and hard to compute.

For the weighted all-pairs shortest path problem (APSP) several algorithms exist and are used in the benchmark. Dijkstra's and Johnson's algorithm have a runtime complexity of $O(ne + n^2 log(n))$, where n is the number of nodes and e the number of edges. Their main difference is, that Johnson's algorithm can additionally deal with negative weights by adjusting weights before searching paths with Dijkstra's algorithm. Since complex network theory forbids negative weights, they have the same complexity (and normally the same practical runtime). Because of their edge dependency, they are suited for sparse networks. In the case of dense networks (ultimately $e \approx n^2 \Rightarrow O(n^3 + n^2 log(n))$), there is to much overhead through jumps in memory, even with adjacency matrices. The iterative Floyd–Warshall algorithm requires $O(n^3)$ (no matter how many edges exist). Since the hidden complexity constant of Floyd–Warshall algorithm is very low, it is mostly faster for rather densely connected networks although it processes non-existing edges [7].

3.2. Libraries involved in the benchmarks

The following libraries were tested:

- *BCT* – The *Brain Connectivity Toolbox* is an extensive complex network library (downloaded: 08.12.2012, [27]) implemented in Matlab and C++. Of course, the C++ version was used for benchmark. To run *BCT* on windows 7, the underlying *GNU Scientific Library* (GSL 1.15, [39]) and *BCT* were compiled with Visual C++ 2010 (see Section 3.2). OpenMP 2.0 support was activated. All calculations were specialized for float values (instead of double or long double) to provide a fair comparison to the other libraries that use fast float values. The *BCT* supports by far the most measures of complex networks of all tested libraries. For both shortest path lengths and clustering coefficients it is referred to as *BCT*.
- *BGL* – The *Boost Graph Library* is a general purpose C++ template graph library (Boost 1.49.0, [40]). For shortest path lengths the Floyd–Warshall algorithm on a *BGL* adjacency matrix was taken as challenger (referred to as *BGL[FLOYD]*). To illustrate an alternative algorithm for rather sparse networks, Johnson's algorithm on a *BGL* adjacency matrix (*BGL[JOHN]*) was included, too. Unfortunately, there is no clustering coefficient algorithm.
- *igraph* – The *igraph* library is specialized in complex network research (version 0.6, [41]). Please note that *igraph* does not support shortest path length algorithms for dense networks (Floyd–Warshall algorithm). For (non-negatively) weighted sparse networks Dijkstra's or Johnson's algorithm can be used. Since both algorithms showed the same runtime performance in pre-tests, Johnson's algorithm was chosen (*IGRAPH[JOHN]*) for a comparison with *BGL[JOHN]*. The undirected clustering coefficients are supported, referred to as *IGRAPH*.
- *networkx* – The only library that was not included in the same benchmark program was the *networkx* library (version 1.7, [42]) for complex networks under Python 2.7.2 that uses routines of *NumPy* 1.6.1 [43]. It supports both measures and is referred to as *NETWORX*.
- *FALCON* – Three configurations will be compared in this benchmark for undirected clustering coefficients and weighted shortest path lengths. That are an optimized CPU implementation on 1 core (*FALCON[CPU1]*), the same approach parallelized on all 4 cores of the test system (*FALCON[CPU4]*) and one optimized implementation on the GPU (*FALCON[GPU]*) with OpenCL.

3.3. Benchmark philosophy

There are many possibilities to test just these two measures (undirected clustering coefficients of all nodes and weighted shortest path lengths of all node pairs) like manipulate the network's number of nodes, number and kind of edges (undirected, weighted or directed) and their distribution. So, one needs to restrict the benchmark to a few important tests.

In this benchmark networks with an arbitrary, fixed number of 5000 nodes were tested for runtime performance of both measures. Every test network was created with the Barábsi–Albert model [44] that generates scale-free random networks that are often observed in nature. This should simulate real-world calculations. To create test networks with an increasing edge density, the parameter that defines a starting network in the Barábsi–Albert model was continuously increased and networks with certain densities were picked out for benchmark. The densities were (approximately) 1%, 2%, ..., 15% (in 1% steps) and 22%, ..., 99% (7% steps). For each edge density there is a

[1] The undirected clustering coefficient of a node is the fraction of all existing triangles out of edges containing this node. The weighted shortest path length of a node pair is the minimal sum of (e.g. inverted) edge weights on a path between this pair.

weighted and an undirected network. This subdivision in little and big density steps is due to the fact that some calculations are such long-winded that only the lower percent range could be included in the diagrams. This network creation method for different densities can only be a very rough approximation of real networks, especially with high densities. In this benchmark networks with an arbitrary, fixed number of 5000 nodes were tested for runtime performance of both measures. Every test network was created with the Barábsi–Albert model [44] that generates scale-free random networks that are often observed in nature. This should simulate real-world calculations. To create test networks with an increasing edge density, the parameter that defines a starting network in the Barábsi–Albert model was continuously increased and networks with certain densities were picked out for benchmark. The densities were (approximately) $1\%, 2\%, \ldots, 15\%$ (in 1% steps) and $22\%, \ldots, 99\%$ (7% steps). For each edge density there is a weighted and an undirected network. This subdivision in little and big density steps is due to the fact that some calculations are such long-winded that only the lower percent range could be included in the diagrams. This network creation method for different densities can only be a very rough approximation of real networks, especially with high densities.

Resulting runtimes of every approach X were measured with millisecond resolution and a speedup factor in respect of a chosen reference approach R (e.g. *BGL[FLOYD]*) was calculated. The speedup factor is $s = \frac{t_R}{t_X}$ with R's runtime t_R and X's runtime t_X. So $s = 2.7$ means that X is 2.7 times faster than R.

Runtime performances of all libraries were tested for stability. Since they did not fluctuate, every calculation was done only once. To prevent errors, every single resulting value was compared to the results of the other libraries (except *NETWORX* that were pretested manually). In case of floating point values a fractional tolerance was used to check for equality of values u and v : $\frac{|u-v|}{|u|} \leqslant \epsilon$and $\frac{|u-v|}{|v|} \leqslant \epsilon$ with float data precision $\epsilon = 10^{-6}$.

To accomplish this automatic error check, the benchmark program had to use a standard format and adapters for ambiguous outputs. For example, when calculating clustering coefficients a possible division by zero is often handled differently by different libraries. These transformations, error checks and initialization of libraries were not part of the runtime measurement, only calculations of measures on their own data format are benchmarked for each library.

3.4. Benchmark hardware and software

In order to achieve a fair comparison under "everyday life" conditions, all libraries were tested in the same program compiled with Microsoft Visual C++ 2010 (-O2, speed optimization) running on Windows 7. One exception was the *networkx* library that was executed with Python 2.7 on this system. Only moderate desktop PC hardware was used: CPU: Intel Core i5 750 (Lynnfield), 4 cores (2.67 GHz), RAM: 4096 MB, DDR3, Dual Channel, latency 7 cycles, GPU: ATI Radeon HD 5770 (Juniper), 10 compute units (850 MHz), 800 shader cores, 1024 MB GDDR5 memory, driver: Catalyst 12.8, OpenCL 1.1 AMD-APP.

3.5. Results for weighted shortest path lengths of all node pairs

Fig. 3 shows runtimes and speedups for calculating the weighted shortest path lengths of all nodes depending on edge densities from tested Barábsi–Albert networks with 5000 nodes. On the left, runtimes of the approaches *IGRAPH[JOHN]*, *BGL[FLOYD]*, *BGL[JOHN]*, *FALCON[CPU1]*, *FALCON[CPU4]* and *FALCON[GPU]* are shown. *NETWORKX* and *BCT* were left out (see below). On the right, speedups of all Floyd–Warshall versions compared with *BGL[FLOYD]* are displayed.

As theoretically expected, there is an almost constant runtime for the Floyd–Warshall versions over all edge densities on the test system. On average, *BGL[FLOYD]* needs 14.5 min, *FALCON[CPU1]* 1.6 min, *FALCON[CPU4]* 1.2 min and *FALCON[GPU]* 25 s. The average speedup factors of *FALCON* in relation to *BGL[FLOYD]* are 9.1× (*FALCON[CPU1]*), 11.8× (*FALCON[CPU4]*) and 34.6× for *FALCON[GPU]*. However *BGL[FLOYD]* shows a runtime increase for very low and very high densities. Since in pre-tests with uniformly distributed edges the *BGL* runtime were as constant as with *FALCON* approaches, this seems to be attributed to the Barábsi–Albert model that fills at first the upper left area of the adjacency matrix with the complete starting network and then adds new edges by preferential attachment. But it is not clear why this imbalanced distribution as the only difference to pre-tests affects the BGL algorithm.

According to theoretical expectations, the runtimes of *BGL[JOHN]* and *IGRAPH[JOHN]* increase with rising edge density. From 1% to 15% density *BGL[JOHN]*'s runtimes rise from 24.7 s to 7.6 min. In this density range *IGRAPH[JOHN]*'s runtimes need from 41.8 s to 12.9 min. Between 1% and 15% the *BGL[JOHN]* is on average 1.7× faster than *IGRAPH[JOHN]*.

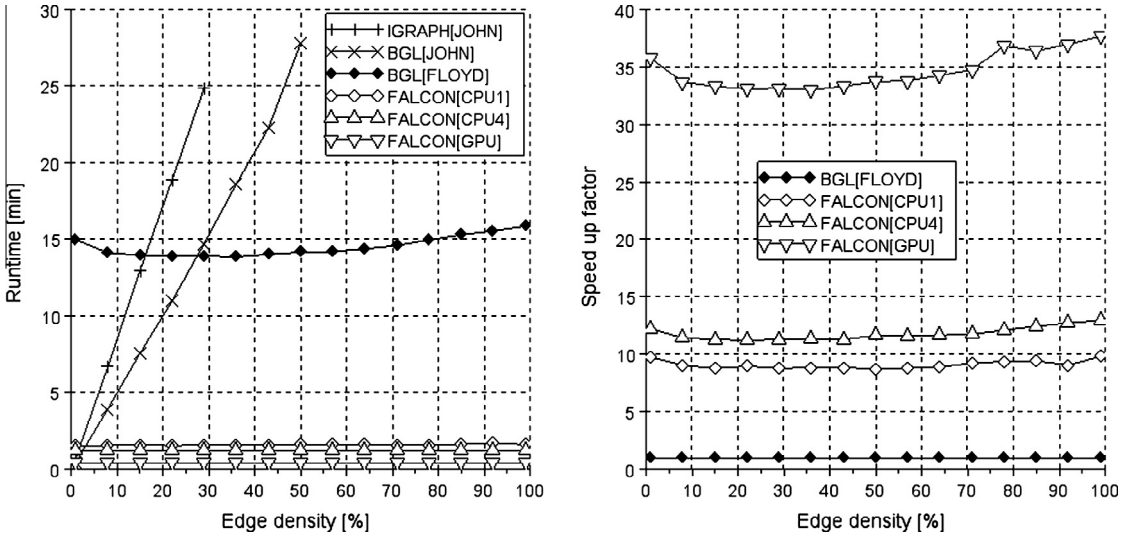

Fig. 3. Calculation performance and speedups of weighted shortest path lengths of all nodes. On the left, runtimes of several libraries are shown for different edge densities. On the right, speedups of all *FALCON* approaches compared to *BGL[FLOYD]* are displayed.

Table 1
FALCON speedups to *IGRAPH[JOHN]*. (FCPU1:*FALCON[CPU1]*, FCPU4: *FALCON[CPU4]*, FGPU: *FALCON[GPU]*.)

Edge density (%)	FCPU1	FCPU4	FGPU
1	0.5	0.6	1.7
2	0.9	1.1	3.1
3	1.4	1.7	5.0
4	2.0	2.5	7.2
5	2.5	3.1	9.1
8	4.3	5.4	15.8
15	8.1	10.4	30.8
22	12.1	15.1	44.6
29	15.7	20.2	59.4

Table 2
FALCON speedups to *BGL[JOHN]*. (FCPU1:*FALCON[CPU1]*, FCPU4: *FALCON[CPU4]*, FGPU: *FALCON[GPU]*.)

Edge density (%)	FCPU1	FCPU4	FGPU
1	0.3	0.3	1.0
2	0.6	0.7	2.0
3	0.9	1.2	3.3
4	1.3	1.6	4.6
5	1.6	2.0	5.7
8	2.5	3.2	9.4
15	4.7	6.1	18.0
22	7.1	8.9	26.1
29	9.3	11.9	35.1

Regarding the edge density dependent *IGRAPH[JOHN]* and *BGL[JOHN]* approach, Tables 1 and 2 show speedups of all three *FALCON* approaches (*FALCON[CPU1]* (FCPU1), *FALCON[CPU4]* (FCPU4) and *FALCON[GPU]* (FGPU)). As expected for density dependent approaches, in very low density ranges (sparse networks) the *IGRAPH[JOHN]* and *BGL[JOHN]* can be faster, leading to speedups below 1.

BCT and *NETWORKX* were left out because they showed very long calculation times for even 1% density. The *BCT* needed 2.3 h. *NETWORKX* needed 36.9 min and is expected to stay at that niveau, since it uses the Floyd–Warshall, too.

3.6. Results for undirected clustering coefficients of all nodes

Fig. 4 shows runtimes dependent on edge density of bechmarked networks with 5000 nodes for calculation of undirected clustering coefficients of all nodes. The runtimes increase as expected with higher density but there are different kinds of ascent. The *NETWORKX* approach shows a very steep ascent that cannot be handled for high densities so only the first development from 1% (15 s) to 13% (16.5 min) is shown for illustration.

All other libraries are displayed in 7% steps from 1% to 99% and rise almost linearly with edge density to maximal runtimes at 99% of 12.6 min (*BCT*), 9.2 min (*IGRAPH*), 1.2 min (*FALCON[CPU1]*), 20.3 s (*FALCON[CPU4]*) and 6.8 s (*FALCON[GPU]*). Over all densities from 1% to 99% the average speedups to *BCT* are 1.5× (*IGRAPH*), 8.9× (*FALCON[CPU1]*), 23.7× (*FALCON[CPU4]*), and 80.7× (*FALCON[GPU]*). Of course, the highest densities are not useful when operating with undirected networks, so it is better to look at the lower and medium density range. Compared to *BCT* and in a range from 1% to 50% the average speedups are 1.6× (*IGRAPH*),7.1× (*FALCON[CPU1]*), 14.7× (*FALCON[CPU4]*), and 56.4× (*FALCON[GPU]*). In the range from 1% to 15% the average speedups are 2.5×, 5.5×, 7.5×, and 40.1×.

3.7. Accuracy and precision

Since only exact algorithms and no approximations or probabilistic approaches were used in this benchmark, all results in every benchmark run were equal to their correspondents in all other included libraries and *FALCON*. Of course, the equality of results is only given by a certain data precision. In this case, float data type had to be used (see Section 2.2). So, accuracy and precision were of course perfect, since there are no false or true negative or false positive results in exact algorithms. A further analysis (e.g. ROC curves, precision-recall) like in many fields of biomedical informatics which deal with uncertain information, probabilistic or approximate algorithms is therefore not meaningful.

Using float weighted networks, but calculating with higher double data precision and afterwards casting back to float had no effect on results like a drift due to rounding. Double data type tests were done with Johnson's algorithm and Floyd–Warshall algorithm in *BGL* and with a double data type version of *BCT*.

4. Discussion

This benchmark reveals a noteworthy advantage of efficient usage of hardware resources, since storage in and access to raw data tables (adjacency matrices) is far more efficient than the

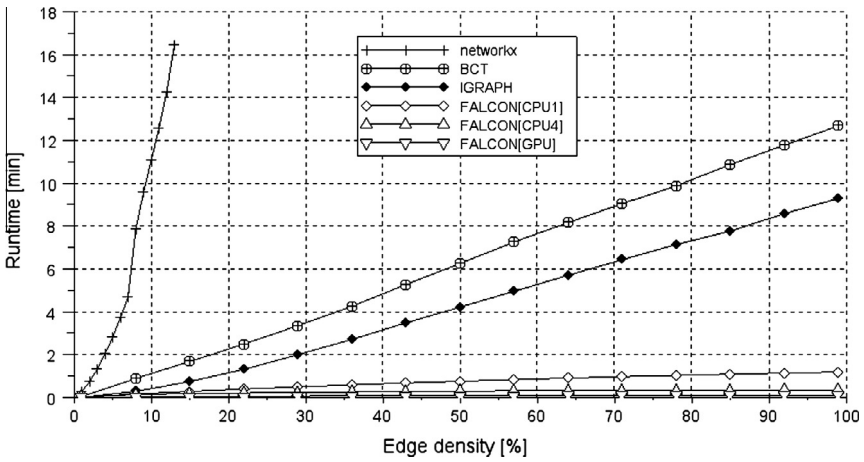

Fig. 4. Calculation performance of undirected clustering coefficients of all nodes. The diagram shows runtimes dependent on edge density of networks with 5000 nodes for the undirected clustering coefficients calculations for all nodes. Please note, that in an unweighted network a density of nearly 100% is senseless. So the medium density range should be taken as "dense". As explained in the text, a comprehensive comparison of weighted versions, in which even a 100% density is able to carry lots of information through different weights, was not possible for a comprehensive benchmark.

(probably) distributed memory access in ordinary adjacency lists. The shortest path lengths and the numbers of triangles (basis of the benchmarked clustering coefficients) are representative for two main routes in the complex network measures tree shown in Fig. 2. If both can be calculated fast, every higher measure benefits from that improvement because all successors can be calculated afterwards in only linear or quadratic time.

Furthermore, this benchmark shows by example, that in practice, not only theoretical runtimes count. The actual implementation of data structures (for example linked lists or matrices) and usage of hardware (e.g. parallelization, cache exploitation, ...) determine the hidden constants in the asymptotic time course as shown especially by the highly parallel GPU approaches. Thus, it is possible that dense network approaches applied on sparse networks are still faster than sparse network algorithms and data structures, as shown here.

Many libraries are designed to process big sparse networks (e.g. *igraph*) or rather small adjacency matrices (e.g. *BCT*). It is important to say, that pure performance is not that essential when investigating only few or small networks. There are much more aspects as, for example, support for many different measures, bindings to easy-to-use languages like Python, R or Matlab and several graph visualizers to make rapid prototyping and network analysis possible. Another case is the universality (e.g. *BGL*), that could prevent specialization to run faster.

In both *FALCON[GPU]* and *FALCON[CPU4]* the Floyd–Warshall-Algorithm is limited by the concurrency of the distributed memory accesses and the need for global synchronization of all processes in its inner loops. Thus, only those inner loops could be optimized. There is potential to search for better access patterns, too.

Going back to our EEG experiment example (see Section 1.3), successive weighted shortest path lengths calculations for all node pairs of 2,000 weighted networks would be done on average in 20.1 days with *BGL[FLOYD]*, 2.2 days (*FALCON[CPU1]*), 1.7 days (*FALCON[CPU4]*) or 14.0 h (*FALCON[GPU]*). The other tested approaches depend on the average edge density of these 2000 networks. If we assume 1% then *BGL[JOHN]* needs 13.8 h and *IGRAPH[JOHN]* 23.2 h. An average density of 5% would result in 5.3 days (*BGL[JOHN]*) and 3.3 days (*IGRAPH[JOHN]*), 15% would cause 10.5 days (*BGL[JOHN]*) and 17.9 days (*IGRAPH[JOHN]*). These examples apply for successive calculations. If there is enough available working memory for more than one network, it is of course possible to start some calculations simultaneously which could ideally speed up the needed time for all networks by the number of parallel CPU cores.

In our demonstrative EEG experiment we looked at weighted clustering coefficients because they contain more information and are far more difficult to compute compared to the undirected version. But to illustrate the advantage of *FALCON* even for undirected edges, let us take a look at Table 3. It shows estimated runtimes for successively calculated undirected clustering coefficients for all nodes of 2000 benchmark networks with each 5000 nodes and varying edge densities. The first column shows a reasonable range of varying edge densities from 1% to 50%. The next columns contain extrapolated runtimes with fixed densities for *BCT*, *igraph* (IG), *FALCON[CPU1]* (FCPU1), *FALCON[CPU4]* (FCPU4) and *FALCON[GPU]* (FGPU). Again, calculating on different networks in parallel could reduce runtime if there is enough working memory.

Table 3
Estimated runtimes for successively calculated undirected clustering coefficients for all nodes for 2000 benchmark networks. (FCPU1:*FALCON[CPU1]*, FCPU4: *FALCON[CPU4]*, FGPU: *FALCON[GPU]*, IG: *igraph*.)

%	BCT	IG	FCPU1	FCPU4 (h)	FGPU (h)
1	1.9 h	0.4 h	0.8 h	0.7	0.4
8	1.2 d	9.6 h	5.4 h	4.8	0.8
15	2.3 d	1.0 d	9.6 h	6.2	1.0
22	3.5 d	1.8 d	13.2 h	7.1	1.1
29	4.7 d	2.8 d	16.6 h	7.9	2.3
36	5.9 d	3.8 d	19.9 h	8.8	2.8
43	7.3 d	4.8 d	22.8 h	9.5	2.9
50	8.7 d	5.9 d	1.1 d	10.1	3.1

Table 4
Working memory or GPU memory needed for adjacency matrix networks with different number of nodes.

Number of nodes	Unweighted	Weighted
1000	0.95 MB	3.8 MB
5000	24 MB	95 MB
10,000	95 MB	381 MB
20,000	381 MB	1.5 GB
30,000	858 MB	3.4 GB
40,000	1.5 GB	6.0 GB
50,000	2.3 GB	9.3 GB

There are a few techniques left that can save both memory and time like using single bits for connections instead of bytes or using a so-called compressed row storage format for sparser networks. Since the optimized implementation of measure calculation is not trivial on those structures, this should be considered for the next version of *FALCON*. Furthermore, GPU calculation is a relatively new and changing field, so it is likely that the described measures can be computed even faster. Table 4 helps to estimate if *FALCON* can be applied (formula from Section 2.3 was used). Please note that the $n \times n$ results of all pairs shortest path length calculations additionally need additional memory besides the network. The density or direction of edges does not matter since adjacency matrices are used. The size of undirected networks could be halved to triangle matrices but that would decrease performance as explained in Section 2.3. To get more data precision, but doubled size, an implementation (and optimization) for the double data type would be useful in the future.

5. Conclusion

This article introduces *FALCON*, a new C/C++ library for dense complex networks. Currently, it computes 12 measures for undirected, weighted or directed networks. It optimizes runtime of every measure, especially the fundamental and long-winded shortest path length of all node pairs and the numbers of triangles around all nodes. This is done at the expense of memory to yield fastest results for especially medium and dense networks. The free working memory limits the maximal number of nodes in networks. A benchmark tested weighted shortest path lengths of all node pairs and the undirected clustering coefficients of all nodes for networks (Barábsi–Albert model) with varying edge density and 5000 nodes. Both measures are hard to compute, fundamental for higher measures and supported by other libraries to enable a fair comparison.

In both cases *FALCON* could save a considerable amount of time compared to other libraries. The reason is that several optimizations like efficient usage of cache memory and SSE are well combinable and lead to much better runtimes, but are restricted to memory consuming adjacency matrices. That is the price of performance in this case, but dense networks should be stored in matrices anyway. Because computing measures for a few single nodes on networks of that size are not a real problem even on desktop computers, *FALCON* calculates every measure for all nodes respectively node pairs at once since this can be done more efficiently and fits the needs of exploratory research.

For the end-user *FALCON* will be compiled into a single executable file (and an OpenCL file) that can be started with command line parameters to handle input files, calculate and generate files

containing results. So, no programming knowledge is required, but nevertheless it can easily be included in other environments. Since command line programs are not user-friendly, a comfortable GUI frontend is planned additionally.

It is not a purpose of *FALCON* to compete with other libraries concerning large and sparse complex networks because it simply cannot handle those ones in this version. The library offers an alternative for extensive calculations on smaller or medium-sized networks with about thousands of nodes. Especially if hundreds or thousands of rather dense networks have to be computed, every performance improvement is worthwhile. The networks do not have to be dense but could be. The higher the density, the more advantageous is *FALCON*. We hope that this new library for fast calculations on complex networks presented here will contribute to explore and analyze new features of dynamically changing neural networks of the brain.

References

[1] Pastor-Satorras R, Vzquez A, Vespignani A. Dynamical and correlation properties of the internet. Phys Rev Lett 2001;87(25):258701.
[2] Newman M. The structure and function of complex networks. SIAM Rev 2003:167–256.
[3] Barrat A, Barthelemy M, Pastor-Satorras R, Vespignani A. The architecture of complex weighted networks. Proc Nat Acad Sci USA 2004;101(11):3747.
[4] Boccaletti S, Latora V, Moreno Y, Chavez M, Hwang D. Complex networks: structure and dynamics. Phys Rep 2006;424(4–5):175–308.
[5] Otte E, Rousseau R. Social network analysis: a powerful strategy, also for the information sciences. J Inform Sci 2002;28(6):441–53.
[6] Pellegrini M, Haynor D, Johnson J. Protein interaction networks. Expert Rev Prot 2004;1(2):239–49.
[7] Cormen T. Algorithmen-Eine Einführung. Oldenbourg Wissenschaftsverlag; 2007.
[8] Nepusz T, Yu H, Paccanaro A. Detecting overlapping protein complexes in protein-protein interaction networks. Nature Meth 2012;9(5):471–2. http://dx.doi.org/10.1038/nmeth.1938.
[9] Goltsev AV, Dorogovtsev SN, Oliveira JG, Mendes JFF. Localization and spreading of diseases in complex networks. Arxiv preprint arXiv:1202.4411.
[10] Gmez S, Gmez-Gardenes J, Moreno Y, Arenas A. Nonperturbative heterogeneous mean-field approach to epidemic spreading in complex networks. Phys Rev E 2011;84(3):036105.
[11] Vidal M, Cusick ME, Barabasi AL. Interactome networks and human disease. Cell 2011;144(6):986–98.
[12] Hartl FU, Bracher A, Hayer-Hartl M. Molecular chaperones in protein folding and proteostasis. Nature 2011;475(7356):324–32.
[13] Serrano M, Bogu M, Sagus F. Uncovering the hidden geometry behind metabolic networks. Mol BioSyst 2012;8(3):843–50.
[14] Sporns O. The human connectome: a complex network. Ann NY Acad Sci 2011;1224(1):109–25.
[15] Bassett DS, Wymbs NF, Porter MA, Mucha PJ, Carlson JM, Grafton ST. Dynamic reconfiguration of human brain networks during learning. Proc Nat Acad Sci 2011;108(18):7641.
[16] Fan Y, Shi F, Smith JK, Lin W, Gilmore JH, Shen D. Brain anatomical networks in early human brain development. Neuroimage 2011;54(3):1862–71.
[17] Greicius MD, Supekar K, Menon V, Dougherty RF. Resting-state functional connectivity reflects structural connectivity in the default mode network. Cereb Cortex 2009;19(1):72–8.
[18] Whitlow CT, Casanova R, Maldjian JA. Effect of resting-state functional MR imaging duration on stability of graph theory metrics of brain network connectivity. Radiology 2011;259(2):516–24.
[19] Moussa MN, Vechlekar CD, Burdette JH, Steen MR, Hugenschmidt CE, Laurienti PJ. Changes in cognitive state alter human functional brain networks. Front Human Neurosci 5.
[20] Tian L, Wang J, Yan C, He Y. Hemisphere-and gender-related differences in small-world brain networks: a resting-state functional MRI study. Neuroimage 2011;54(1):191–202.
[21] Yan C, Gong G, Wang J, Wang D, Liu D, Zhu C, et al. Sex-and brain size-related small-world structural cortical networks in young adults: a DTI tractography study. Cereb Cortex 2011;21(2):449–58.
[22] de Haan W, van der Flier WM, Wang H, Van Mieghem PF, Scheltens P, Stam CJ. Disruption of functional brain networks in alzheimer's disease: what can we learn from graph spectral analysis of resting-state magnetoencephalography? Brain Connect 2012;2(2):45–55. http://dx.doi.org/10.1089/brain.2011.0043.
[23] Wang L, Zhu C, He Y, Zang Y, Cao QJ, Zhang H, et al. Altered small-world brain functional networks in children with attention-deficit/hyperactivity disorder. Human Brain Mapp 2009;30(2):638–49.
[24] Yu Q, Sui J, Rachakonda S, He H, Gruner W, Pearlson G, et al. Altered topological properties of functional network connectivity in schizophrenia during resting state: a small-world brain network study. PLoS One 2011;6(9):e25423.
[25] Bernhardt BC, Chen Z, He Y, Evans AC, Bernasconi N. Graph-theoretical analysis reveals disrupted small-world organization of cortical thickness correlation networks in temporal lobe epilepsy. Cereb Cortex 2011;21(9):2147–57.
[26] Menon V. Large-scale brain networks and psychopathology: a unifying triple network model. Trends Cognit Sci 2011;15(10):483–506.
[27] Rubinov M, Sporns O. Complex network measures of brain connectivity: uses and interpretations. Neuroimage 2010;52(3):1059–69.
[28] McIntosh AR. Contexts and catalysts. Neuroinformatics 2004;2(2):175–81.
[29] Brandes U. A faster algorithm for betweenness centrality. J Math Sociol 2001;25(2):163–77.
[30] Brandes U. On variants of shortest-path betweenness centrality and their generic computation. Social Networks 2008;30(2):136–45.
[31] AMD. OpenCL™ zone | AMD developer central <http://developer.amd.com/zones/OpenCLZone>; 2012.
[32] NVIDIA, OpenCL | NVIDIA developer zone <http://developer.nvidia.com/opencl>; 2012.
[33] Harish P, Narayanan P. Accelerating large graph algorithms on the GPU using CUDA. High Perform Comput – HiPC 2007 2007:197–208.
[34] Katz G, Kider Jr J. All-pairs shortest-paths for large graphs on the GPU. In: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on graphics hardware; 2008. p. 47–55.
[35] Fog A. Optimizing software in C++: an optimization guide for windows, linux and mac platforms.
[36] Drepper U. What every programmer should know about memory <http://people.redhat.com/drepper/cpumemory.pdf>.
[37] Saramki J, Kivel M, Onnela JP, Kaski K, Kertesz J. Generalizations of the clustering coefficient to weighted complex networks. Phys Rev E 2007;75(2):027105.
[38] Onnela J, Saramki J, Kertsz J, Kaski K. Intensity and coherence of motifs in weighted complex networks. Phys Rev E 2005;71(6):065103.
[39] GSL, gnu.org <http://www.gnu.org/software/gsl/> (June 2012).
[40] Boost. The boost graph library – boost 1.49.0 <http://www.boost.org>; 2012.
[41] Csardi G, Nepusz T. The igraph software package for complex network research. InterJournal Comp Syst 1695. <http://igraph.sourceforge.net/>.
[42] Hagberg A, Swart P, Chult DS. Exploring network structure, dynamics, and function using NetworkX. Tech. rep., Los Alamos National Laboratory (LANL); 2008.
[43] Numpy. Scientific computing tools for python numpy <http://numpy.scipy.org/>; 2012.
[44] Albert R, Barabsi A. Statistical mechanics of complex networks. Rev Mod Phys 2002;74(1):47.